A compiler for parallel Unity programs using OpenMp

Raphaël Couturier, Bertrand Couturier, Dominique Méry

A compiler for parallel Unity programs using OpenMp

Raphaël Couturier (speaker) Bertrand Couturier
Dominique Méry
UMR n^o 7503 LORIA
Université Henri Poincaré Nancy 1
Campus Scientifique
BP 239 54506 Vandœuvre-lès-Nancy (France)
email: couturie,couturib,mery@loria.fr

Abstract ***In this paper we explain how we built a compiler to transform a specification of a parallel Unity program into parallel Fortran code executable on a shared memory SGI machine using OpenMp. We use the Unity version modified by Eric F Van de Velde with a few adaptations. As non-determinism is not allowed with this version and as the mapping onto the shared memory machine is simply done using compiler directives to parallelize matrix and vector operations, the results show us that we can write very high level parallel programs and execute them efficiently.***

Keywords: Unity, OpenMP, compiler

1 Introduction

The design of parallel programs has been greatly influenced by the UNITY framework introduced by Chandy and Misra [1] and the success of UNITY is mainly due to the simplicity of its programming notation, the integration of a programming logic (a fragment of linear temporal logic) and the relationship between the programming notation and different kinds of architectures. The current work reported in this paper addresses the question of translation of a variant of Unity, called parallel Unity [2], into parallel Fortran executed on a shared memory SGI machine using OpenMp. As we are producing Fortran code for a parallel machine, namely the origin2000, we should study the efficiency of the produced programs whilst maintaining the "correctness" of the initial Unity program. A main benefit from preserving the correctness of the abstract notation in the produced Fortran code is the simplicity of the notation. This allowed us to evaluate the use of the translator in the training of physics and chemistry students of our university: first of all, they can easily use the abstract notation during the specification phase and, secondly, the produced Fortran code can be improved by themselves.

Van Velde [2] introduced the parallel Unity programming notation and carried out many case studies; the first work was to define the programming language and to build a syntax analyzer. Translating by hand, we obtained a Fortran version of the case studies of Van Velde and we studied the efficiency of the produced code. Finally, a general compiler from parallel Unity to Fortran OpenMp was written in Java, and a Java interface allows testing of the code resulting from a parallel Unity program.

Work reported in this document contributes to the introduction of formal methods in the development cycle of highly performance programs. As we have carried out our research very closely with users of the highly performant programming facilities (provided by the Origin 2000 of the Charles Hermite Center) we hope to convince users and students that an abstract notation facilitates the development

of Fortran code and can help in the debugging phase.

The paper is organized as follows. Section 2 describes the parallel Unity programming notation. Section 3 introduces OpenMP and the technique of parallelization using it. Section 4 shows how we have defined the mapping from parallel Unity into Fortran. In section 5, we analyze results related to the efficiency of the code produced. Finally, we conclude our work.

2 Which version of Unity

Unity [1] was developed to provide a general framework in which we can define programs, properties and mappings, within a uniform notation. The initial structure of a Unity program consists of:

- a *declaration* section used to define variables,
- an *always* section in which we can define certain program variables as functions of other variables,
- an *initially* section to initialize variables and
- an *assign* section containing a set of assignment statements.

The assignment section carries out the main role of a Unity program since all computations are realized within it. Assignment can be simple or multiple, for example:

$x, y, z := 0, 1, 2$

corresponds to:

$x := 0 \;||\; y := 1 \;||\; z := 2$, where the operator $||$ expresses the parallelism between statements.

Quantifications are allowed and this is simpler than enumeration (these statements are also called quantified statements), for example:

$< ||i : 0 \leq i \leq N :: A[i] := B[i] >$

corresponds to:

$A[0] := B[0] \;||\; .. \;||\; A[i] := B[i] \;||\; .. \;||\; A[N-1] := B[N-1]$

Assignment can be guarded, thus $a := b$ if $a > b$ will execute the assignment $a := b$ only if the expression $a > b$ is true. Guards offer the possibility to express non-determinism in a program. This is an advantage if we reason on an abstract program but, contrastingly, this introduces complications if we want to build efficient code.

In his book [2], Eric Van de Velde explains that good notations help to reduce problems of parallel program development. But, of course, they do not eliminate all problems. He chooses to write his programs with a modified version of Unity in which non-determinism is not allowed. Thus, the mapping onto a high performance machine could be made more efficient.

For convenience, he added the following instruction:

$t :=< +m : 0 \leq m \leq M :: x[m] >$

which performs the summation of $x[m]$. The operator $+$ can be replaced by the operator $*$ and we can quickly compute a product. Trying to parse programs written by Van de Velde, we had some problems with the grammar, and thus we chose to modify it slightly. We summarise the differences from the grammar of Van de Velde.

- We do not allow multiple assignments, since multiple assignments on scalar variables are inefficient when there are few assignments (these could be interesting with matrix operations but, in such a case, the parallelization of these operations is generally more powerful).
- All operations written using a "mathematical form" are represented in a "computer science form". Thus, $\sqrt{a}$ is written $sqrt(a)$, x^T is written $transp(x)$ (this is the transposition of a vector). All multiplications are explicit, thus Ax is written

$A * x$ and x^2 is written $x * x$, for simplification. The construction of a complex $a + ib$ is written $cmplx(a, b)$. $\overline{a}$ is written $conjg(a)$. The imaginary part of a is written $imag(a)$ and the real part of a is $real(a)$.

3 Parallelization with OpenMP

OpenMp is a new standard for building parallel programs using compiler directives. It was defined to use Scalable Shared Memory multiprocessors, SSMP, as efficiently as possible. On such a machine, as the memory is shared by all the processors, one processor can load or store directly any shared address. And, as the programmer can also specify that a variable is private to a processor, this system provides a very simple yet powerful model for expressing and managing parallelism in an application [3]. Furthermore, since directives can be ignored, the same code can also be compiled into a sequential program. This is very useful because the transition between the sequential and the parallel program is very easy — one has just to compile the program in the required mode.

The execution model of the OpenMP API [4] is defined as following. First, a program written in OpenMP Fortran begins execution as a single process, called the master thread of execution. Then the master process runs sequentially until it reaches any one of the two main parallel constructs, either of which can use the *fine grained* paradigm and parallelize most of the loops in a code (sometimes loops are so big that the term *fine grained* is perhaps not appropriate), or one can use the *coarse grained* paradigm by defining sections in which the code is executed by multiple threads (this is quite similar to the concept of parallel programs using a communication library since each process runs the same program and executes according to its processor number). Thus, when a parallel construct is reached, the master thread gives work to each thread, it synchronizes all the threads in the team and it keeps running sequentially after the execution of the parallel construct.

Through our experience with parallelization on a shared memory machine [5], we think that the parallelization of loops is simpler than the parallelization of large sections of code for two reasons. Firstly the parallelization of loops can be done incrementally. We start with a loop and we manage a second one only after having well parallelized the first. Secondly, the parallelization of one loop requires only local analysis of this loop, whereas in the second case we must globally analyse the structure of all the sections — this work is far from being a simple task when the code is large and when we parallelize it without (or with very little) assistance from the initial developers of the code.

To parallelize a loop, the user must first ensure that the loop is parallelizable, i.e. all the iterations in the loop can be executed in any order and always give the same result. With OpenMP, the task for us is to change loops in which such a transformation is not possible; we give some examples, in the figure 1, to clarify this property.

- In *loop 1*, each iteration `i` has a value `k` used to compute the nested loop on `j` which sets up the value of `B(i)`. Thus, each iteration is independent from the others; consequently, all iterations can be run concurrently.
- In *loop 2*, each iteration `i` has a value `k` used to compute the value `B(k)` in the nested loop on `j`. We can distinguish two cases depending on the values of `A`. If we know that each element of `A` is unique and different from the others, all iterations can be run concurrently, but if this is not the case, we cannot execute concurrently all iterations and have the same result as in the sequential case.
- In *loop 3*, each iteration `i` has a value `k` which is used to set up the value of `B(j)` in the nested loop `j`. Thus, some elements of `B` are set up by most of the iterations `i` and consequently we cannot run concurrent iterations on `i`.

```
DO i=1,n
  k=A(i)
  DO j=1,k
    B(i)=B(i)+C(i)
  ENDDO
ENDDO
```

loop 1

```
DO i=1,n
  k=A(i)
  DO j=1,k
    B(k)=B(k)+C(k)
  ENDDO
ENDDO
```

loop 2

```
DO i=1,n
  k=A(i)
  DO j=1,k
    B(j)=B(j)+C(j)
  ENDDO
ENDDO
```

loop 3

Figure 1: 3 different loops which can or cannot be run concurrently

If each iteration in a loop is independent then the user must specify the status of each variable. In OpenMP, there are 3 different clauses to define the status of a variable. The clause `PRIVATE` means that the variable is private to a thread. A variable involved in a loop execution (the variable `i` in our examples) is always `PRIVATE` since each iteration of the loop works with a different value of the variable. More generally, each variable which has a scope local to an iteration is a `PRIVATE` variable. Even though the scope of a variable in Fortran cannot be syntactically local to a loop, it takes only a glance to see if a variable is used before and after a required loop.

Scalar variables on which we perform the same partial operation at each iteration are `REDUCTION` variables. If a variable has this status this means that each thread has its local copy of the variable. Thus each thread computes a partial result of the variable and after the loop the operation is applied on each local variable in order to give the correct result. For example, if we have a variable involved in a summation, each thread computes a partial sum with the operator plus, and, after the loop, all partial results are gathered with the operator plus used to give the sum. Unfortunately this technique cannot be used with arrays and we must therefore simulate this. If reduction is applied on a real variable, we sometimes obtain results which are a little different from the sequential result — this is caused by the error of approximation but this problem of diverging results is seldom serious.

The last possible status for a variable is `SHARED`. Such a variable is shared among all the threads, so we must ensure that either the variable is only read or that it is modified by no more than one thread simultaneously. In general if a `SHARED` variable is scalar then the variable is only read and if it is a vector then we must ensure that each element is not modified concurrently.

The status of the variables involved in the previous loops of figure 1 is shown in figure 2. We saw previously that *loop3* cannot be parallelized and that is why we cannot give its parallel version!

To summarise, if we want to parallelize a loop, either we are lucky and we just have to give the status of each variable or we must modify [1] the code and specify the status of all the variables involved in the loop thus obtaining the same result in the parallel case as seen in the sequential case.

4 Implementation

To execute Unity programs, we produce Fortran 77 code with compiler directives available with OpenMP and then we compile it on the SGI Origin 2000. The translation from Unity code into Fortran is realized with the help of the Java Parser Generator [6]. This tool is very simple to use and we define the grammar (an

[1] Most of the time, the modifications are very minor.

```
!$OMP  PARALLEL DO PRIVATE(i,k,j)
!$OMP+ SHARED(A,B,C)
DO i=1,n
  k=A(i)
  DO j=1,k
    B(i)=B(i)+C(i)
  ENDDO
ENDDO
```
loop 1
```
!$OMP  PARALLEL DO PRIVATE(i,k,j)
!$OMP+ SHARED(A,B,C)
DO i=1,n
  k=A(i)
  DO j=1,k
    B(k)=B(k)+C(k)
  ENDDO
ENDDO
```
loop 2

Figure 2: loop 1 and loop 2 parallelized (loop 2 is parallelizable only if all elements of A are distinct)

LL(1) by default but it can be an LL(k) in some parts) by constructing relations between each term of the grammar and then the parser generator builds skeletons of each Java class. Then we must specify for each class how to perform the translation. The most difficult work consists of generating code to correctly handle matrix operations. We review the work done for each section of a unity program.

The declaration section is just the translation of a Unity type into a Fortran type. We give an example in figure 3.

Unity
```
lamb : complex;
v : array [100] of complex;
A :  array [100 X 100] of real;
tau : real
```
Fortran
```
Complex lamb
Complex v(0:100)
Real A(0:100,0:100)
Real tau
```

Figure 3: The translation of a Unity declaration into Fortran

The *initially* section is similar to the *assign* section. If we consider scalar initialization, or vector or matrix initialization, the code produced is the same as for the *assign* section. In fact, we distinguish both these sections only in order to separate them, that's why the syntax is slightly different but the semantics is the same. In some cases, the *initially* section is suppressed because we need to assign random values and this kind of operation must be done in the assign section.

In the assign section, we have to handle 4 kinds of statements: pure assignment statements, loop statements, conditional statements and reduction statements.

In a pure assignment section, we must distinguish simple assignment from vector or matrix assignments: in the second case we have to capture possible errors on these operations and automatically generate the code. Thus, simple assignments are directly translated and we add subroutines in the Fortran code to handle all cases of matrix and vector operations (multiplication of matrix-matrix, matrix-vector, scalar-vector, scalar-matrix, scalar product of vectors). As an example, we give in figure 4 the matrix-vector product subroutine.

```
      Subroutine MVMult
     *  (M,V,size_i,size_j,VRes)
      Integer size_i,
     *  size_j,i,j,k
      Real M(0:size_i,0:size_j),
     *  V(0:size_j),VRes(0:size_i),
     *  Sum

!$OMP PARALLEL DO PRIVATE(i,j,k,Sum)
      Do i=0,size_i
        Do j=0,size_j
          Sum=0
          Do k=0,size_j
            Sum=Sum+M(i,k)*V(k)
          Enddo
          VRes(i)=Sum
        Enddo
      Enddo
      End
```

Figure 4: matrix-vector product

Loops (or quantified statements) are translated easily and we have only to specify the status of each variable, which can be done automatically. Since Unity considers that the operator || means that statements in a loop can be executed in parallel, we do not suppose that a loop is unparallelizable (we consider that if the operator || is used then the user knows that the loop is parallelizable). We handle this case by generating a compiler directive that enables the loop to be executed in parallel. Quantified assignments with the sequential operator ";" are done sequentially without any directive. We give an example of a parallel loop:

```
<|| j : i+1 <= j < n ::
       ar[i,j] := ar[i+1,j]>
```

and its translation in Fortran:

```
!$OMP PARALLEL DO PRIVATE(j)
        Do j = i+1 , n-1
          ar(i,j) = ar(i+1,j)
        Enddo
```

To simplify the use of our translator, we built an interface in Java, which allows us to open a file, edit it and save it. If we consider that our Unity file is suitable, we can try to translate it into a Fortran file. If the translation succeeds, we can then execute the program on the machine. A dialog box allows us to choose the option for the Fortran compiler, or select the browser to see the "help", or to view the number of processors used for each execution. After a translation the user can switch between the Unity program, the Fortran program and the execution view. Figure 5 shows a screen shot of the tool.

5 Results

All the experiments were done on the Origin 2000 while sharing the load with other users. The consequence is that depending on the load of the machine, we could have results with a few variation in time of execution. For all the following results the time is expressed in seconds.

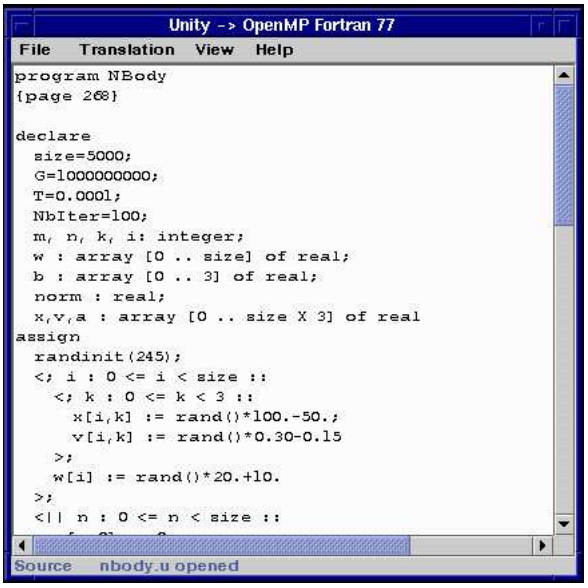

Figure 5: Unity to Fortran translator interface

The N-Bodies problem with 5000 bodies and 100 iterations (all bodies are interacting)

number of processors	time	speed up
1	980	1
4	252	3.88
16	69	14.20

The Gauss Seidel resolution with a 4000*4000 matrix

number of processors	time	speed up
1	6644	1
4	1857	2.50
16	741	8.96

The Poisson solver with a 6000*3000 matrix and 400 iterations

number of processors	time	speed up
1	7217	1
4	2020	3.57
16	647	11.15

For these the examples, we obtain globally satisfying results (bearing in mind that the code automatically produced can be optimized again by hand, if it is necessary). We show the code of the Gauss Seidel resolution, to provide

an idea of the structure.

```
program GaussSeidel
declare
  i, j, k : integer;
  n = 4000;
  ar : array[n+1 X n] of real
assign
  initialization of the matrix
  randinit(234);
  <; i : 0 <= i < n+1 ::
    <; j : 0 <= j < n ::
         ar[i,j]:=rand()*10000+1
    >
  >;
  triangularization
  <; i : 0 <= i < n-1 ::
    <|| j : i+1 <= j < n+1 ::
      <; k : i+1 <= k < n ::
         ar[j,k] := (ar[j,k] -
           (ar[i,k]*ar[j,i]) / ar[i,i])
      >
    >;
    <|| j : i+1 <= j < n :: ar[i,j] := 0 >
  >;
  resolution
  <; i : n > i >= 0 ::
    ar[i,i] := ar[i+1,i]/ar[i,i];
    <|| j : 0 <= j < i ::
      ar[i,j] := ar[i+1,j]-ar[i,j]*ar[i,i]
    >;
    <|| j : i+1 <= j < n ::
      ar[i,j] := ar[i+1,j]
    >
  >;
  <; i : 0 <= i < n ::
    print ar[0,i]
  >
end
```

6 Concluding remarks and future work

The parallel Unity programming notation is more abstract than the produced Fortran and it results in more readable code that can be derived in a formal way using the Unity methodology. The simplicity is inherent to Unity because it relates programming notations to proofs. In future work, we plan to connect our tool to a proof assistant, based on PVS[7], to allow us to derive properties on Unity notations.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation.* Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[2] Eric F. Van de Velde. *Concurrent scientific computing.* Springer-Verlag, 1994.

[3] White paper: OpenMP: A proposed standard api for shared memory programming. see http://www.openmp.org/.

[4] OpenMP fortran application program interface, October 1997. see http://www.openmp.org/.

[5] Raphaël Couturier and Dominique Méry. Parallelization of a Monte Carlo simulation of a spins system. In Hamid R. Arabnia, editor, *Parallel and Distributed Processing Techniques and Applications - PDPTA'98, Las Vegas, USA*, July 1998.

[6] Sun Microsystems. http://www.suntest.com/JavaCC/.

[7] Sam Owre, Natarajan Shankar, and John Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.